# 1.ACID

**ACID，是指在[数据库](http://lib.csdn.net/base/14" \o "MySQL知识库" \t "_blank)管理系统（DBMS）中，事务(transaction)所具有的四个特性：原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation，又称独立性)、持久性(Durability)。**

**在数据库系统中，一个事务是指：由一系列数据库操作组成的一个完整的逻辑过程。例如银行转帐，从原账户扣除金额，以及向目标账户添加金额，这两个数据库操作的总和，构成一个完整的逻辑过程，不可拆分。这个过程被称为一个事务，具有ACID特性。**

**1)原子性(Atomicity)**

**一个事务(transaction)中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。**事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

**2)一致性(Consistency)**

**事务的一致性指的是在一个事务执行之前和执行之后数据库都必须处于一致性状态。如果事务成功地完成，那么系统中所有变化将正确地应用，系统处于有效状态。如果在事务中出现错误，那么系统中的所有变化将自动地回滚，系统返回到原始状态。**

**3)隔离性(Isolation)**

**指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看到中间状态的数据。**

**4)持久性(Durability)**

**指的是只要事务成功结束，它对数据库所做的更新就必须永久保存下来。即使发生系统崩溃，重新启动数据库系统后，数据库还能恢复到事务成功结束时的状态。**

    事务的（ACID）特性是由关系数据库管理系统（RDBMS，数据库系统）来实现的。**数据库管理系统采用日志来保证事务的原子性、一致性和持久性。日志记录了事务对数据库所做的更新，如果某个事务在执行过程中发生错误，就可以根据日志，撤销事务对数据库已做的更新，使数据库退回到执行事务前的初始状态。**

# 2.CAP

**一致性(Consistency)、可用性(Availability)和分区容忍性(Partitiontolerance)。CAP原理指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。**这是Brewer教授于2000年提出的，后人也论证了CAP理论的正确性。

**1)一致性（Consistency） ：(** 同样数据在分布式系统中所有地方都是被复制成相同。**)**

**对于分布式的存储系统，一个数据往往会存在多份。简单的说，一致性会让客户对数据的修改操作（增/删/改），要么在所有的数据副本（replica）全部成功，要么全部失败。即，修改操作对于一份数据的所有副本（整个系统）而言，是原子（atomic）的操作。**

**如果一个存储系统可以保证一致性，那么则客户读写的数据完全可以保证是最新的。不会发生两个不同的客户端在不同的存储节点中读取到不同副本的情况。**

**2) 可用性（Availability） ：(** 所有在分布式系统活跃的节点都能够处理操作且能响应查询。**)**

**可用性很简单，顾名思义，就是指在客户端想要访问数据的时候，可以得到响应。但是注意，系统可用（Available）并不代表存储系统所有节点提供的数据是一致的。这种情况，我们仍然说系统是可用的。往往我们会对不同的应用设定一个最长响应时间，超过这个响应时间的服务我们仍然称之为不可用的。**

**3) 分区容忍性（Partition Tolerance） ：(** 分区容错性: 在两个复制系统之间，如果发生了计划之外的网络连接问题，对于这种情况，有一套容错性设计来保证。**)**

**如果你的存储系统只运行在一个节点上，要么系统整个崩溃，要么全部运行良好。一旦针对同一服务的存储系统分布到了多个节点后，整个存储系统就存在分区的可能性。比如，两个存储节点之间联通的网络断开（无论长时间或者短暂的），就形成了分区。一般来讲，为了提高服务质量，同一份数据放置在不同城市非常正常的。因此节点之间形成分区也很正常。**

    Gilbert 和Lynch将分区容忍性定义如下：除全部网络节点全部故障以外，所有子节点集合的故障都不允许导致整个系统不正确响应。即使部分的组件不可用，施加的操作也可以完成。

     一个数据存储系统不可能同时满足上述三个特性，只能同时满足其两个特性，也就是： CA,CP,AP。可以这么说，当前所有的数据存储解决方案，都可以归类的上述三种类型。

**CA  满足数据的一致性和高可用性，但没有可扩展性，如传统的关系型数据，基本上满足是这个解决方案，如ORACLE , MYSQL 的单节点，满足数据的一致性和高可用性。**

**CP  满足数据的一致性和分区性，如Oracle RAC ，Sybase 集群。虽然Oracle RAC具备一点的扩展性，但当节点达到一定数目时，性能（也即可用性）就会下降很快，并且节点之间的网络开销还在，需要实时同步各节点之间的数据。**

**AP 在性能和可扩展性方面表现不错，但在数据一致性方面会用牺牲，各节点的之间数据同步没有哪么快，但能保存数据的最终一致性。当前热炒的NOSQL大多类是典型的AP类型数据库。**

综合上述，架构师不要企图设计一套同是满足CAP三方面的数据库。只能在根据业务场景，对数据存储要求有所折衷。

# 3.BASE

**接受最终一致性的理论支撑是BASE模型，BASE全称是BasicallyAvailable（基本可用）, Soft-state（软状态/柔性事务）, Eventually Consistent（最终一致性）。BASE模型在理论逻辑上是相反于ACID（原子性Atomicity、一致性Consistency、隔离性Isolation、持久性Durability）模型的概念，它牺牲高一致性，获得可用性和分区容忍性。**

**最终一致性：**

**最终一致性是指：经过一段时间以后，更新的数据会到达系统中的所有相关节点。这段时间就被称之为最终一致性的时间窗口**

**CAP和ACID一致性区别**

**ACID一致性是有关数据库规则，如果数据表结构定义一个字段值是唯一的，那么一致性系统将解决所有操作中导致这个字段值非唯一性的情况，如果带有一个外键的一行记录被删除，那么其外键相关记录也应该被删除，这就是ACID一致性意思。**

**CAP理论的一致性是保证同样一个数据在所有不同服务器上的拷贝都是相同的，这是一种逻辑保证，而不是物理，因为光速限制，在不同服务器上这种复制是需要时间的，集群通过阻止客户端查看不同节点上还未同步的数据维持逻辑视图。**

**CAP原理指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。因此在进行分布式**[**架构**](http://lib.csdn.net/base/16)**设计时，必须做出取舍。而对于分布式数 据系统，分区容忍性是基本要求 ，否则就失去了价值。因此设计分布式数据系统，就是在一致性和可用性之间取一个平衡。对于大多数web应 用，其实并不需要强一致性，因此牺牲一致性而换取高可用性，是目前多数分布式**[**数据库**](http://lib.csdn.net/base/14)**产品的方向。(**一般情况下CAP理论认为你不能拥有上述三种中两种，这是一个实践总结：**当有网络分区情况下，也就是分布式系统中，你不能又要有完美一致性和100%的可用性，只能这两者选择一个。在单机系统中，你则需要在一致性和延迟性latency之间权衡)**

**当然，牺牲一致性，并不是完全不管数据的一致性，否则数据是混乱的，那么系统可用性再高分布式再好也没有了价值。牺牲一致性，只是不再要求关系型数 据库中的强一致性，而是只要系统能达到最终一致性即可，考虑到客户体验，这个最终一致的时间窗口，要尽可能的对用户透明，也就是需要保障“用户感知到的一致性”。**通常是通过数据的多份异步复制来实现系统的高可用和数据的最终一致性的，“用户感知到的一致性”的时间窗口则 取决于数据复制到一致状态的时间。

**最终一致性(eventually consistent)**

     对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多并发访问时更新过的数据如何获取的问题。从服务端来看，则 是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

**从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性 。如果能容忍后续的部分或者全部访问不到，则是弱一致性 。 如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。**

    最终一致性根据更新数据后各进程访问到数据的时间和方式的不同，又可以区分为：

* **因果一致性** 。如果进程A通知进程B它已更新了一个数据项，那么进程B的后续访问将返回更新后的值，且一次写入 将保证取代前一次写入。与进程A无因果关系的进程C的访问遵守一般的最终一致性规则。
* **“读己之所写（read-your-writes）”一致性** 。当进程A自己更新一个数据项之后，它总是访问到 更新过的值，绝不会看到旧值。这是因果一致性模型的一个特例。
* **会话（Session）一致性** 。这是上一个模型的实用版本，它把访问存储系统的进程放到会话的上下文中。只要 会话还存在，系统就保证“读己之所写”一致性。如果由于某些失败情形令会话终止，就要建立新的会话，而且系统的保证不会延续到新的会话。
* **单调（Monotonic）读一致性** 。如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那 个值之前的值。
* **单调写一致性** 。系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一致性，就非常难以编程 了。

    上述最终一致性的不同方式可以进行组合，例如单调读一致性和读己之所写一致性就可以组合实现。并且从实践的角度来看，这两者的组合，读取自己更新的数据，和一旦读取到最新的版本不会再读取旧版本，对于此架构上的程序开发来说，会少很多额外的烦恼。

     从服务端角度，如何尽快将更新后的数据分布到整个系统，降低达到最终一致性的时间窗口，是提高系统的可用度和用户体验非常重要的方面。对于分布式数 据系统：

* N — 数据复制的份数
* W — 更新数据是需要保证写完成的节点数
* R — 读取数据的时候需要读取的节点数

**如果W+R>N，写的节点和读的节点重叠，则是强一致性。例如对于典型的一主一备同步复制的关系型数据库，N=2,W=2,R=1，则不管读 的是主库还是备库的数据，都是一致的。**

**如果W+R<=N，则是弱一致性。例如对于一主一备异步复制的关系型数据库，N=2,W=1,R=1，则如果读的是备库，就可能无法读取主库 已经更新过的数据，所以是弱一致性。**

对于分布式系统，为了保证高可用性，一般设置N>=3。不同的N,W,R组合，是在可用性和一致性之间取一个平衡，以适应不同的应用场景。

* **如果N=W,R=1，任何一个写节点失效，都会导致写失败，因此可用性会降低，但是由于数据分布的N个节点是同步写入的，因此可以保证强一致性。**
* **如果N=R,W=1，只需要一个节点写入成功即可，写性能和可用性都比较高。但是读取其他节点的进程可能不能获取更新后的数据，因此是弱一致性**。这种情况 下，如果W<(N+1)/2，并且写入的节点不重叠的话，则会存在写冲突